Vienna University of Technology
Information Systems Institute
Distributed Systems Group

Distributed Continuous Data Aggregation Over Web Service Event Streams

Under Review for Publication in -

W. Hummer, B. Satzger, P. Leitner, S. Dustdar
hummer@infosys.tuwien.ac.at

TUV-1841-2011-4 6/21/11

We present a distributed platform for continuous event-based aggregation of Web services and data. The platform both actively monitors Web services and receives XML events using WS-Eventing. An aggregation is composed of multiple inputs such as Web service invocations and event subscriptions, which are formulated and processed in a query language based on XQuery. Our query model allows to specify data dependencies among the inputs of an aggregation. We use an XQuery language extension that enables users to easily express which data are exchanged between the individual inputs. The dependencies are automatically resolved and continuously updated with the required data at execution time. The query specification is abstracted from its physical distribution, and the platform is able to distribute the execution among multiple computing nodes. We evaluate several aspects of the implemented prototype in a Cloud Computing setting.

Keywords: WS-Aggregation, Complex Event Processing, Data Aggregation, Continuous Query, Event-Based System

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
http://www.infosys.tuwien.ac.at/

Distributed Continuous Data Aggregation Over Web Service Event Streams

Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Austria
Email: {lastname}@infosys.tuwien.ac.at

***Abstract*—We present a distributed platform for continuous event-based aggregation of Web services and data. The platform both actively monitors Web services and receives XML events using WS-Eventing. An aggregation is composed of multiple inputs such as Web service invocations and event subscriptions, which are formulated and processed in a query language based on XQuery. Our query model allows to specify data dependencies among the inputs of an aggregation. We use an XQuery language extension that enables users to easily express which data are exchanged between the individual inputs. The dependencies are automatically resolved and continuously updated with the required data at execution time. The query specification is abstracted from its physical distribution, and the platform is able to distribute the execution among multiple computing nodes. We evaluate several aspects of the implemented prototype in a Cloud Computing setting.**

I. Introduction

Throughout the last years, developments around the World Wide Web have been coined by a move from an Internet of (static) documents to an Internet of services [1] with a vivid exchange of invocation messages and events. The paradigm of Service-oriented Computing [2] considers services as the building blocks for distributed applications. While the classical Web model is a client-server and request-response model, more and more emphasis is put on loosely coupled distributed systems, asynchronous processing and event-driven architectures [3]. This trend is underpinned by the popularity of Web services and well-adopted standards like WS-Eventing [4].

Today, the Extensible Markup Language (XML) is the primary format for data exchange on the Web. The XML query language XQuery [5] provides a powerful means to arbitrarily extract, transform and generate XML content. The current working draft of XQuery version 3.0, among other things, now supports queries over (possibly infinite) sequences of XML nodes. This feature has been largely influenced by Botan et al. [6] who proposed to extend XQuery with *window functions*. These window functions (or window *clauses*) can provide the basis for Complex Event Processing (CEP) [7] when applied to XML event streams, i.e., sequences of single temporally decoupled XML messages sent from an event producer to a consumer. On top of XQuery, however, the event subscription, collection, correlation and propagation still need to be tailor-made and require custom implementation on the application level. This inherent complexity becomes even harder to handle when there are dependencies between individual event streams, and if one event subscription (i.e., request of a consumer to initiate an event stream) needs to be updated when another event stream produces a certain result or pattern.

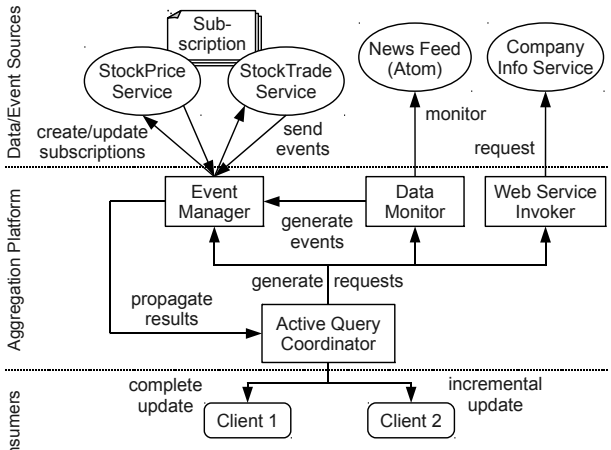
In a previous paper, we have introduced the *WS-Aggregation* framework [8], a scalable platform which allows to combine and process data from heterogeneous Web resources using configurable distribution strategies. Whereas our previous work focused on synchronous, stateless aggregation of static Web data, we now extend the approach to support active (continuous) queries over dynamically changing data and event streams. The platform employs an novel active query model for event-based data aggregation which not only processes events from a single source or subscription, but actively creates and renews event subscriptions based on user-defined data dependencies and generation templates. An aggregation query consists of an arbitrary number of inputs from Web services and allows to specify data dependencies between any two inputs i_1 and i_2. When i_1 yields a new result, the invocation or event subscription for i_2 is generated and renewed. An important aspect is that the logical (or functional) query specification is abstracted from its physical distribution, which allows to partition the execution on multiple computing nodes for deployment in a scalable Cloud Computing environment.

The remainder of this paper is structured as follows. In Section II we first introduce a scenario of event-based data aggregation, and Section III gives some background information about XQuery window clauses. The main part in Section IV presents the general query model and the eventing model, and discusses aspects of distributed query execution and runtime query updates. Section V details the platform implementation, and Section VI evaluates different aspects of the solution. Section VII covers a discussion of related work, and Section VIII concludes the paper with an outlook for future work.

II. Scenario

We introduce a motivating scenario taken from the domain of financial computing and stocks trading, in which Web services provide information about companies and live data on stock prices and trades. Fig. 1 sketches a high-level approach which pursues the goal of consuming events and retrieving data from the services, in order to combine and aggregate the information, and to construct an XML document that is actively updated when the underlying data change.

We distinguish three basic types of receiving data from the sources: (1) the *StockPrice* and *StockTrade* services allow clients to subscribe for certain events which are then sent



Stock	Company	News	Price	Status	Max. Bids
GM	General Motors	GM is...	35.21	↑ rising	1.8 Mio.
F	Ford Motor C.	-	15.23	~	-
TM	Toyota Motor C.	-	45.05	~	-

Fig. 1. Event-Based Continuous Data Aggregation Scenario

using WS-Eventing, (2) the *Atom*-based [9] *News* syndication feed is regularly monitored for changes, (3) the *Company* information service contains static data that rarely changes and is requested only once without being updated. The consuming clients specify the query and receive the aggregated data in the form of an XML document. Depending on the capabilities of the client, either the whole document is always transmitted (complete update) or the client receives only incremental updates and is responsible to construct the final document.

The aggregation platform mediates between the data providers and the data consumers, and coordinates the query execution. From a high-level perspective, the required core components are an Event Manager (EM), a Data Monitor (DM), a Web Service Invoker (SI) and an Active Query Coordinator (AQC). The EM is responsible for managing subscriptions with the target services and for receiving events from it. The DM repeatedly retrieves data from the monitored resource and generates an event to report any changes. The collected events and results from EM, DM and SI are fed into the AQC, whose responsibility is to update all dependencies and generate new requests as needed. In that sense, the AQC operates in a loop of stepwise and continuous execution of the aggregation query. Note that the aggregation platform may have to be distributed over several computing nodes, be it for performance reasons or due to higher-level constraints (e.g., the StockPrice and StockTrade services should report to physically separated endpoints). However, from the client's viewpoint the platform should appear as a single entity.

We assume that the XML document resulting from the data aggregation should contain a table with the current stock price of all stock symbols that have been recorded, together with general information about the company. Furthermore, the result should indicate when a stock shows three or more consecutive price rises, in which case the largest bid volume (amount of stocks offered by a trader) is displayed. Additionally, if the platform detects that a stock has risen recently *and* that traders are placing big volume bids (e.g., ≥ 1 million), the table should display live news about the respective companies.

A. Interdependent Event Streams

Fig. 2 illustrates two sample event streams of the StockPrice and StockTrade services. Initially, only a subscription for StockPrice exists, and the service continuously sends stock price ticks as events. As soon as three consecutive rises are detected for *"GM"* (events printed in bold), the Event Manager requests a new event subscription with the StockTrade service to receive the volume (amount) of all *bids* and *asks* for the *"GM"* placed on the market. Finally, this subscription gets destroyed when five consecutive ticks (also in bold text) are below the last price of the rising sequence (35.7).

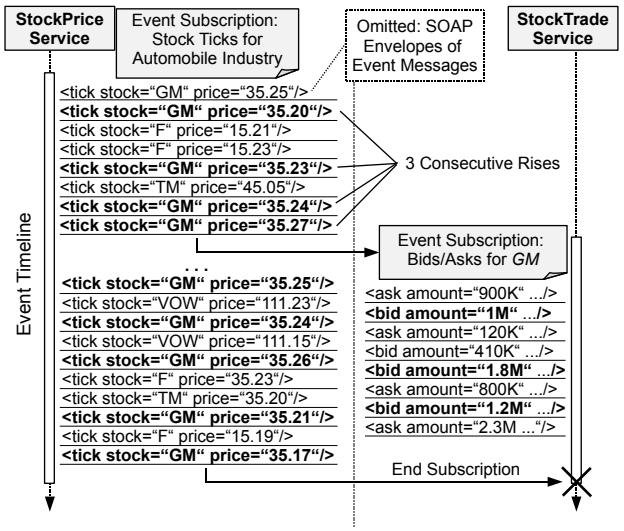


Fig. 2. Sample Event Streams and Lifecycle of Event Subscriptions

We observe that the presented scenario poses several challenges to CEP. The basic requirement is the possibility to detect patterns in event streams in order to trigger new actions. This is a core research topic in CEP, and we can build on the existing work (e.g., [10], [11]). In this paper, we utilize XQuery to detect patterns in single streams, and perform distributed execution of continuous queries to aggregate data from multiple sources. We thereby focus particularly on dependencies between event streams which determine the data flow and the lifecycle of event subscriptions.

III. Background

We now provide some background information on window clauses in XQuery, which is essential for understanding the remainder of the paper. More concretely, the current draft of XQuery 3.0 introduces *tumbling window* and *sliding window* clauses which we utilize as the basis for CEP over streams of XML elements. An exemplary window query operating on the events of our StockTrade service is printed in Listing 1. The predefined variable `$input` points to the sequence of events that have been received by the platform so far.

The query in Listing 1 creates a *window* `$w` (sequence of consecutive items drawn from the sequence in `$input`) for each subsequence of `$input` for which the *start* and *end* conditions apply. The start condition is that the bid amount is at least *1000000*, and the end condition is that a higher amount is found than in the previous sequence (window). In other words, the query splits the total sequence `$input` into chunks of consecutive event sequences (stored in the loop variable `$w`), and returns end item of each subsequence (which is greater than the end item of the preceding subsequence).

```
1  for tumbling window $w in $input/bid
2  start $s at $spos previous $sprev when
3     number($s/@amount) ge 1000000
4  end $e next $enext when
5     $spos le 1 or
6     number($sprev/@amount) lt number($e/@amount)
7  return
8  <maxbid stock="{$e/@stock}">{$e/@amount}</maxbid>
```

Listing 1. XQuery Window Clause for Events from StockTrade Service

The slightly more complex query in Listing 2 uses a sliding window to find consecutive price rises in the stock ticks. Using a function `x:sorted` (lines 1-4), the clause in lines 5-12 filters windows with rising prices for all ticks with the same symbol as the start item (`$s`) of the window. The returned `rising` element (line 13) is then further processed by the platform.

```
1  declare function x:sorted($list as item()*) as boolean{
2       (every $i in 1 to (count($list) - 1) satisfies
3       number($list[$i]) lt number($list[$i + 1]))
4  };
5  for sliding window $w in $input/stock
6  start $s at $spos when true()
7  end $e at $epos when count(
8    $input/stock[position() = $spos to $epos]
9                [@symbol eq $s/@symbol]
10   ) ge 3
11 let $ticks := $w[@symbol eq $s/@symbol]
12 where x:sorted($ticks)
13 return <rising symbol="{$s/@symbol}">{$ticks}</rising>
```

Listing 2. XQuery Window Clause for Events from StockPrice Service

The defining characteristic of tumbling windows is that new windows are only created when the previous window has been closed, whereas sliding windows may overlap [6]. The two types of window clauses provide orthogonal functionalities, and it depends on the application which of the two is to be used. Note that XQuery provides these powerful language features for queries over event streams, but does not deliver explicit means for continuous queries from multiple sources, which consider data dependencies between event streams, automatically request and retrieve data from services, and manage the lifecycle of event subscriptions. How this is achieved by WS-Aggregation is further discussed in Section IV.

IV. Event-Based Web Data Aggregation

In this section we detail our approach for event-based querying and aggregation of Web services and data. Using the terminology of **aaS* (*Everything as a Service* [12]), WS-Aggregation offers "Aggregation as a service" (AaaS) in the sense that it constitutes a self-contained platform with an invokable Web service interface, which, upon request, performs event-based data aggregation tasks on behalf of the requesting clients. In the following we discuss both the functional aspects (i.e., how can clients express aggregation queries), and the non-functional aspects which affect the platform's internal structure and mode of operation. We first discuss the model that is used to construct aggregation queries in Section IV-A. The actual query language is based on XQuery and introduces additional language constructs that are tailored to the query model used. The query model is abstracted from distribution aspects and allows for parallel and distributed processing of the requests. Processing of XML data in general and execution of XML queries with XQuery in particular are processor- and memory-intensive tasks. This is even more true for the continuous and active queries studied in this work, which require the current query execution state to be cached in main memory or at least persisted on a hard drive for later retrieval. With this in mind, the WS-Aggregation platform is particularly designed for scalability and Cloud-based deployment, which will be discussed in more detail in Section IV-C.

A. Query Model

To initiate a query execution, users formulate an aggregation query that contains all necessary information to retrieve and process the events and data. A simplified version of the query model is illustrated as a UML class diagram in Fig. 3. The model builds on our previous work [8] and extends it with eventing-specific aspects. The central entity *AggregationQuery* specifies the EndpointReference (EPR) [13] used to receive result updates (*notifyTo*). An aggregation query contains multiple *Input*s (identified by *ID*) that determine how data from external sources are retrieved and inserted into the active query.

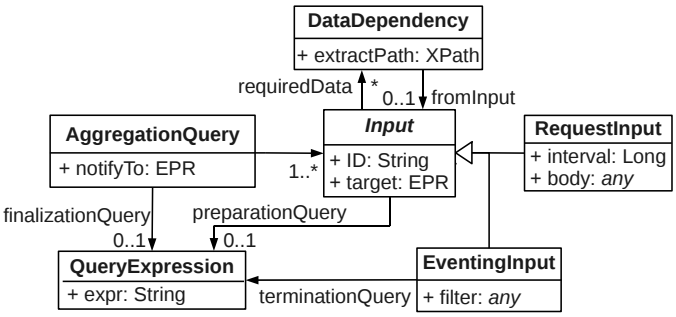


Fig. 3. Query Model for Continuous Event-Based Web Data Aggregation

The *EventingInput* entity creates event subscriptions with an optional *filter* that is evaluated by the target Web service as defined in WS-Addressing. On the other hand, *RequestInput* is used for single documents retrieved in a request-response manner. In both cases, the *target* EPR specifies the location of the service. The optional *interval* attribute allows to continuously monitor a Web service or document for changes. As soon as a change is detected, an internal event is generated as described in the scenario in Section II and in Section IV-D.

Besides input entities, an aggregation query contains XQuery-based *QueryExpression*s. A *preparationQuery* expression may be used to prepare and transform the result of an Input immediately when it arrives at the platform. In the case

of a RequestInput, the preparation query performs a one-time data transformation (e.g., extracting the news for only a certain company from the scenario News Feed), whereas to "prepare" an EventingInput a window query is continuously executed on the event stream to yield new results. To specify the condition for ending an event subscription, an EventingInput is associated with a *terminationQuery*. When this query yields a boolean *true* result, the target service is automatically invoked with a WS-Eventing *Unsubscribe* message to destroy the subscription. Finally, the *finalizationQuery* combines all the prepared results and constructs the final output document.

B. Input Data Dependencies

A core feature in the query model is the concept of data dependencies between two inputs i_1 and i_2, which signifies that i_2 can only be "activated" if certain data from i_1 are available and can be inserted into i_2. Activation in this context means that the input becomes usable for the active query only when all data dependencies are resolved. The query model in Fig. 3 associates an Input (*receiving* input), via the association class *DataDependency*, with an arbitrary number of required data from other Inputs (*providing* inputs) of the same AggregationQuery. The attribute *extractPath* is an XPath which points to the data in the providing input. If the optional association *fromInput* is set, the data will be extracted from a specific Input; otherwise, if *fromInput* is unknown, the platform continuously matches *extractPath* against the available inputs and extracts data when this XPath evaluates to *true*.

```
[new] DataDependency ::= "$" Name? "{" PathExpr "}"
[new] EscapeDollar ::= "$$"

[125] PrimaryExpr ::= DataDependency | Literal |
       VarRef | ParenthesizedExpr | OrderedExpr |
       ContextItemExpr | FunctionItemExpr |
       FunctionCall | UnorderedExpr | Constructor
[145] CommonContent ::= DataDependency |
       EscapeDollar | PredefinedEntityRef |
       "{{" | "}}" | CharRef | EnclosedExpr
[204] ElementContentChar ::= Char − [{}<&$]
[205] QuotAttrContentChar ::= Char − ["{}<&$]
[206] AposAttrContentChar ::= Char − ['{}<&$]
```

Listing 3. XQuery Language Extension for Data Dependencies

We propose an extension to the XQuery language to account for simple modeling of data dependencies as studied in this paper. The modifications are printed in EBNF (Extended Backus-Naur Form) syntax in Listing 3. The new construct is named *DataDependency* and consists of a dollar sign ("$"), an optional *Name* token referencing the *ID* of the providing input, and an XPath expression (*PathExpr*) specifying the *extractPath* in curly brackets ("{", "}"). To express that a string "`${foo}`" should be interpreted as a verbatim string and not as a data dependency, a double dollar sign (*EscapeDollar*) is used for escaping ("`$${foo}`"). The *DataDependency* token is added to the definition of *PrimaryExpr* (rule 125 in the current version of XQuery 3.0) and *CommonContent* (rule 145). Furthermore, to satisfy parser consistency of the syntax rules, the single dollar sign needs to be appended to the list of "exceptional" (non-content) characters (rules 204 to 206). Section V provides further implementation details.

1) Scenario Query Model: Based on the four inputs for the scenario services (SP = StockPrice, ST = StockTrade, CN = Company News, CI = Company Information), the expressions in Fig. 4 illustrate how data dependencies (highlighted in bold font) can be utilized in different parts of an aggregation query, including the WS-Eventing *filter* for the ST subscriptions (input 2), the *preparationQuery* for the monitored CN Feed (input 3), and the SOAP body of the CI request (input 4). The *terminationQuery* of ST is similar to its *preparationQuery* (see Listing 1) and not printed in the figure. The input results are directly accessible in the *finalizationQuery*, e.g., `//news` inserts the results of input 3 into the final result. The figure also depicts the *Dependency Graph* indicating which input expects data from which other inputs (CI and ST from SP, CN from ST). This graph is automatically constructed from the aggregation query and checked for circular dependencies.

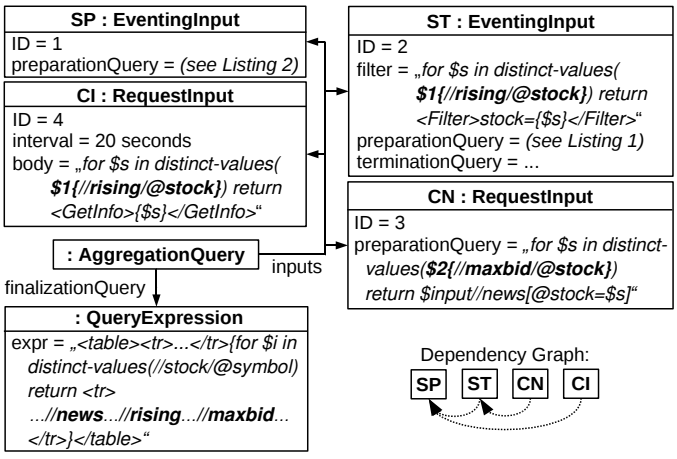


Fig. 4. Aggregation Query for Data Aggregation Scenario

It is important to notice that the XQuery expressions are evaluated in a preprocessing step which generates the list of actual elements to be used as, e.g., *filter* for ST and *body* for CI. For instance, if during evaluation of `$1{//rising/@stock}` the platform extracts three stock symbols ('GM','F','TM') from the prepared result of input 1, three instances of input 2 are generated for each of the corresponding filter expressions. Analogously, three instances of input 4 are generated with corresponding `GetInfo` service requests. Similarly, the *preparationQuery* of CN loops over all `maxbid` stock symbols from input 2 (cf. preparation query in Listing 1) and outputs all news lines specific to these symbols.

C. Distributed Query Execution

To serve a large number of user requests and multiple active queries simultaneously, the platform employs a scalable distributed processing model with several loosely coupled nodes working collaboratively. In addition to the obvious performance reasons, query distribution may also be required or desired from a higher-level (business) perspective. For instance, the data may have to be physically separated according

to business policies. Moreover, if multiple data sources are spread over a large geographical distance, the aggregation can be organized in a location-based hierarchical structure, e.g., with regional and national nodes (for more details see [8]).

Algorithm 1 Processing of Active Query with Dependencies

$results \leftarrow$ new result store // variable for aggregation results
function generateRequests(AggregationQuery r)
1: **while** r contains independent inputs **do**
2: $I \leftarrow$ determine independent inputs in r
3: **for all** $i \in I$ **do**
4: $G \leftarrow$ generate actual inputs from i using XQuery engine
5: **for all** $input \in G$ **do**
6: $aggr \leftarrow$ determine aggregator to handle $input$
7: **if** $aggr$ is self **then**
8: $result \leftarrow$ invoke $input$ on $input.target$
9: $result \leftarrow$ apply preparation query to $result$
10: add $result$ to $results$, update dependencies
11: **else**
12: delegate request with $input$ to $aggr$
13: **end if**
14: **end for**
15: **end for**
16: **end while**
function onEvent(Event e) // called by WS-Eventing service
1: add e to event buffer of e
2: **for all** EventingInput i affected by e **do**
3: $result \leftarrow$ apply preparation query of i to event buffer of e
4: add $result$ to $results$, notify clients, update dependencies
5: generateRequests($i.aggregationQuery$) // issue new requests
6: **end for**

The fundamental assumption of WS-Aggregation is therefore that multiple aggregator machines collaboratively process the queries and events requested by the clients. The set of available aggregators is stored in a central service registry, which allows to dynamically select a subset of aggregators responsible for executing each individual query. The overall request is then split up into smaller ("atomic") parts that can be processed by a single node (*generateRequests* function in Algorithm 1). It should be stated that the single parts cannot be regarded as completely isolated units, because, according to the query model, there often exist data dependencies between them. Each time a new event is received and added to the result store (*onEvent* function), the dependencies are updated and possibly new request inputs are generated. Note that several inputs, possibly from different aggregation queries, can be affected by an event in the *onEvent* function.

Line 6 in the function *generateRequests* indicates that a responsible aggregator is determined for each input. WS-Aggregation considers different configurable distribution strategies, and allows to either specify fixed input-to-aggregator mappings, or to assign inputs automatically. In the latter case, the platform has the possibility to perform load balancing: in general, new inputs are assigned to aggregators with the lowest load (CPU, memory, number of active queries). A second important distribution goal for event-based processing is to bundle inputs with the same underlying event stream. Consider any two inputs i_1 and i_2 which receive the same ticks from the StockPrice service, but use a different preparation query to filter certain information. If these two inputs are handled by some aggregator a, both queries can use a shared event buffer and redundancies are avoided to save memory. Of course, this approach does not scale infinitely, hence new inputs are assigned to a new aggregator if the load of a reaches a certain threshold. The evaluation in Section VI further discusses this aspect.

D. Eventing Model and Runtime Query Update

The core achievement of WS-Aggregation is the provision of a general-purpose service for event-based aggregation of Web data in a scalable and self-adapting distributed platform. The highly dynamic system has to deal with temporally decoupled events and frequent updates which affect different aspects (e.g., data dependencies, monitoring) of active query execution. In the following we briefly discuss the different types of externally and internally generated events, and how these events are processed in the platform.

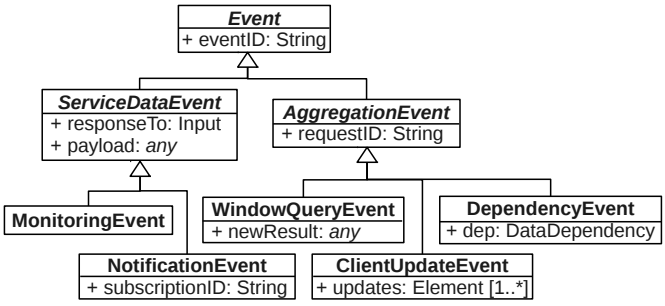


Fig. 5. Eventing Model

The diagram in Fig. 5 classifies the events in a type hierarchy with the common base class *Event*. The *ServiceDataEvent* class represents external events that are generated either by WS-Eventing enabled sources directly (*NotificationEvent*) or as a result of the platform's monitoring activity (*MonitoringEvent*). Each instance of *AggregationEvent* represents an internally generated event that either yields a new result from evaluating a window query on an event stream (*WindowQueryEvent*), or causes an update of the final document provided to the client (*ClientUpdateEvent*). The purpose of *DependencyEvent* is to update the dependency graph when an aggregator a_1 reports that some result it has received from another aggregator a_2 (as part of the result propagation) fulfills a dependency in one of the inputs managed by a_1. This event type only plays a role for data dependencies which contain the XPath expression but no specific ID of the input from which the data should be extracted.

V. Implementation

This section discusses the implementation of the presented approach. WS-Aggregation employs multiple aggregator nodes which collaboratively implement the functionality of the aggregation platform as sketched earlier in Fig. 1. From an external viewpoint, an aggregator is solely defined by its Web

service interfaces. Therefore, the internal structure depicted in Fig. 6 can be seen as a prototype or guideline, and specialized implementations of aggregator nodes can be plugged into the platform, as long as the required service interfaces are implemented (concerning both structural and functional aspects) and the aggregator registers itself in the *Service Registry*. The WS-Eventing compliant *Eventing Interface* is used to receive XML events from the data providing services. An *Event Store* buffers and forwards the XML events to the *Query Engine* which consists of the *Preprocessor* (responsible for processing the XQuery data dependency extensions) and the third-party (hence depicted in gray) *XQuery Engine* (we currently use *MXQuery*[1]). The *Active Query Coordinator* (AQC), accessible from the *Aggregation Interface*, creates and maintains aggregation queries, determines which data dependencies are fulfilled and which new inputs can be activated.

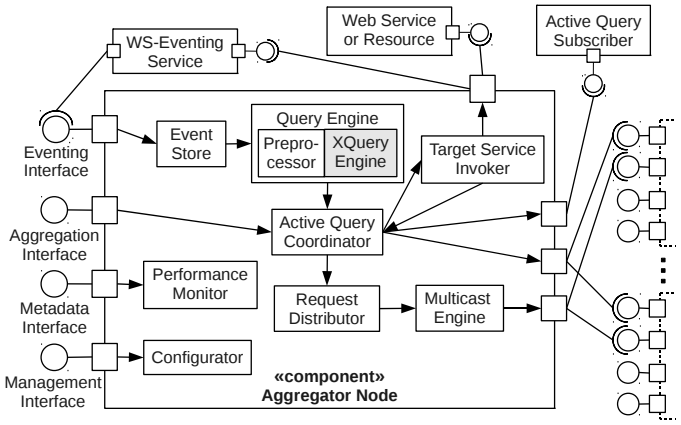


Fig. 6. Core Components and Connectors of Aggregator Nodes

The AQC forwards activated inputs to the *Request Distributor*, which implements configurable query distribution strategies. Moreover, the AQC uses the Eventing Interface of partner aggregators to propagate *WindowQueryEvent*s. To communicate with other nodes, the Request Distributor makes use of the *Multicast Engine*, which contacts the Aggregation Interface (to delegate the execution of inputs) or *Metadata Interface* of the partners. Results are pushed to the clients (Active Query Subscribers) by the AQC; alternatively, clients can poll for results (e.g., useful for Web browsers). Incremental updates, which transmit only the new/changed parts of the final result, are implemented using VMSystem's *XML diff*[2] tool.

The implementation of the Preprocessor and the XQuery language extension for data dependencies is based on JavaCC[3], a parser generator for Java. The EBNF syntax rules of XQuery were extended with the modifications in Listing 3 and transformed into a format readable for JavaCC. The parser generated by JavaCC reads in the extended XQuery expressions and creates an in-memory representation (abstract syntax tree), which is used to extract the data dependencies.

[1]http://mxquery.org/

[2]http://www.vmsystems.net/vmtools/

[3]https://javacc.dev.java.net/

VI. Evaluation

This section covers an evaluation of different aspects of event-based data aggregation in WS-Aggregation. We first discuss the proposed XQuery extension for data dependencies in Section VI-A. To assess the end-to-end performance of the framework, we have set up a comprehensive test environment in the Amazon Elastic Compute Cloud[4] (EC2).

A. XQuery Language Extension for Data Dependencies

In the following we briefly evaluate the proposed XQuery extension for data dependencies and put it into perspective with alternative approaches, focusing on existing standards for XML querying and processing. Note that this evaluation should merely give an idea of the expressiveness of the proposed language extension, but cannot give a complete overview of the (basically unlimited) alternatives.

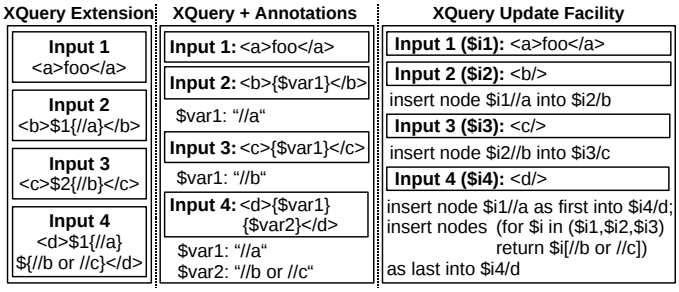


Fig. 8. Possible Alternatives for Expressing Data Dependencies

Fig. 8 illustrates an exemplary aggregation query with four inputs and various dependencies (e.g., Input 2 depends on Input 1, Input 4 depends on data from any Input matching `//b or //c`). The left part of the figure depicts how these dependencies are expressed in WS-Aggregation. A possible alternative would be to use standard XQuery in which the positions of required data are indicated using predefined external variables (e.g., `$var1`, `$var2`, ..). The user would then have to attach annotations to the input expressions, which specify the XPath tests for the target data. In contrast to our approach which inserts the data directly into the source (e.g., replaces `"${//a}"` with `"<a>foo</a>"` in Input 2), the annotations approach requires (1) to set the value(s) of external variables in the XQuery engine, and (2) to execute the engine to receive the final result. A third conceivable method to accomplish data dependencies is the use of the XQuery Update Facility [14], which allows to apply transformations (e.g., insert, update, delete) to existing XML markup. The advantage of this approach is that the original input contains no additional tags, but XQuery Update statements are arguably harder to write for users. A further drawback of this approach is that the XPath describing the data dependency is not directly accessible. This means that, unless the XPaths are specified as an additional information, the update has to be executed each time the value returned from some input changes - regardless of whether the required data are available or not.

[4]http://aws.amazon.com/ec2/

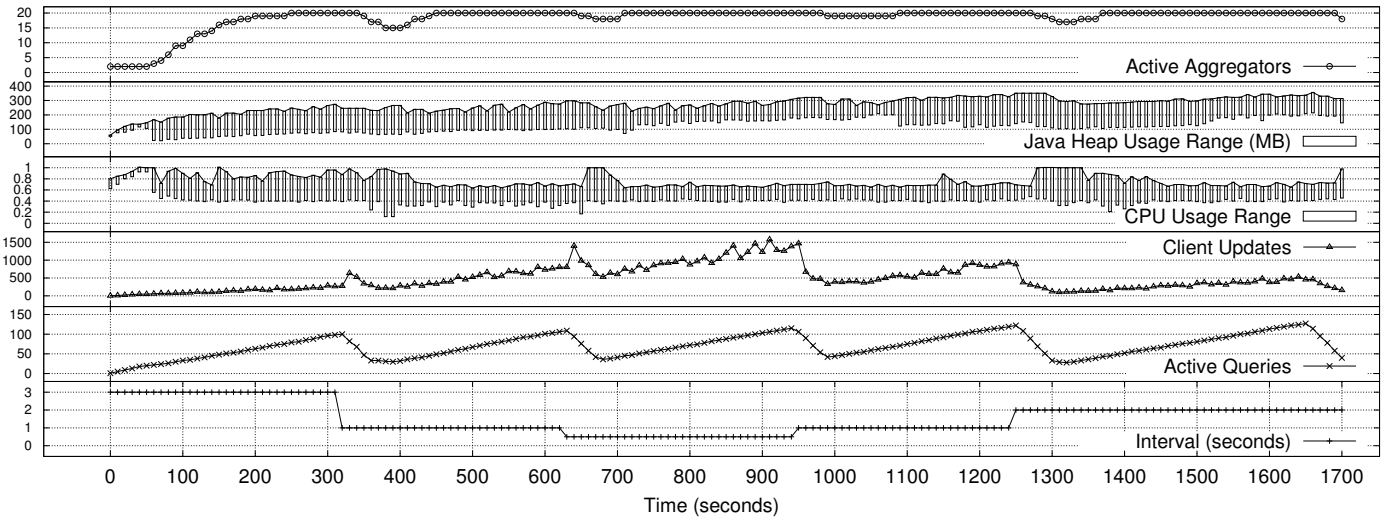


Fig. 7. Performance Results of Executing Multiple Simultaneous Scenario Aggregation Queries with Different Event Frequencies

B. End-To-End Framework Performance

To evaluate the runtime performance of the framework, we have launched an initial number of 15 small virtual machine instances in EC2. We then deployed 15 aggregator nodes, one node per instance. During execution, the framework was configured to deploy up to 5 additional instances, which is achieved using the Web services based API of EC2. Furthermore, we deployed the four Web services which provide randomized test data for the functionalities of the scenario (see Section II), and a *gateway* service whose responsibility is to act as a central entry point for clients and to select master aggregators which coordinate a query's execution.

Fig. 7 illustrates the results of the scenario execution. The x-axis shows the time in seconds. The lowermost part of the figure plots the interval in which the test StockPrice and StockTrade services publish events to the platform. Over time, various test clients deployed in a LAN outside of EC2 (average latency of 60ms) have requested and terminated multiple (up to 125 simultaneous) executions of the scenario aggregation in different variants (sub-plot *Active Queries*). The number of *Client Updates* per ten seconds (up to 1500 around time point 900) is largely influenced by a combination of event *Interval* (between 0.5 and 3 sec.) and Active Queries, and also depends on the random test data and the state of each active query.

The framework monitors the resource consumption using the Java Management Extensions (JMX). Heap memory and CPU are shown in Fig. 7 with both the range (minimum to maximum) and the trendline of the maximum over all active aggregators. The platform heuristically attempts to distribute the total load, based on CPU/memory usage, active aggregators and active queries. Up to second 50, the queries are handled by only two aggregators, because as discussed in Section IV-C, the aim is to assign queries for one event stream to the same aggregator. However, after 60 seconds, additional aggregators are involved to avoid performance deterioration due to the increasing load. When ten aggregators are active, the platform requests new machines in addition to the 15 initial instances. The startup time (roughly 40 sec.) includes EC2 overhead and the time to start, initialize and add the aggregator to the registry. As the active queries decrease, some aggregators become idle (e.g., time 400); the timeout for releasing unused EC2 resources is configurable and should be at least a couple of minutes because aggregators may become used again (e.g., time 400-500) and EC2 instances are billed per hour.

We observe that the load is stable and equally distributed (small CPU and memory ranges) when the number of active queries only slightly changes (e.g., between time 400-600 or 700-900); however, rapid changes in the active queries cause load peaks (e.g., seconds 300-400, 600-700, 1250-1350), as event stores are initialized/terminated and many objects need to be allocated/freed. For space constraints, the figure does not include the aspect of adding multiple Active Query Subscribers (clients) to an existing query, but our results indicate that pushing out additional notifications has only a small performance impact (compared to adding new queries).

Note that the memory consumption grows particularly at the beginning, because the aggregators perform internal caching, both implicitly in the underlying frameworks (e.g., in the Jetty HTTP server) and explicitly in the business logic (e.g., aggregator endpoints queried from the registry are cached for a limited amount of time). A factor that evidently raises memory management issues is the need to store past events for evaluation of window functions. Fortunately, MXQuery employs sophisticated algorithms to free unused items in the input buffer, and we have run several complex queries with up to 1 million events without memory leaks. Upon destruction of an active query, all resources and stored events are freed.

VII. Related Work

The broad range of applications for event processing has spurred the interest of both industry and research for this topic [15]. Important topics in CEP include query rewriting [16], pattern matching over event streams [10], aggregation

of events [17] or event specification [18]. In the context of service-oriented computing, some work on bringing the power of CEP into service environments has been carried out within the VRESCo project [19]. However, so far eventing is mostly used for monitoring (e.g., [20], [21]), while WS-Aggregation uses the CEP notion for service-based data aggregation.

Related to the data aggregation aspect is the *Qizx* XML database engine [22], which provides an XQuery implementation supporting most 3.0 features. As WS-Aggregation, Qizx Server performs data management tasks on demand, however, it lacks support for complex event processing and WS-Eventing. Similarly, the *Active XML* project [23] provides distributed data management and different styles of data integration. The core idea is to enrich XML documents with embedded instructions for Web service invocations. However, ActiveXML follows a top-down approach (the complete "template" for the target document is used as the starting point), whereas WS-Aggregation employs a bottom-up query model, in which atomic inputs of an aggregation query are executed and correlated to dynamically compose the final result.

Another related problem is distributed filtering of XML documents, as for instance researched in XFilter [24]. XFilter is an XPath based approach to structural filtering of documents. A contribution in a similar direction has recently been presented in [25], where nondeterministic finite automata have been distributed over Pastry distributed hashtables in order to filter XML in a distributed way. Both approaches have a strong focus on high-performance XML filtering, while WS-Aggregation provides a much larger feature set. Other approaches that consider distributed processing of event streams generally sacrifice query expressiveness and Web standards for performance. For instance, S4 [26] is a distributed stream computing platform that encodes events as key/value pairs. The keys are used to map events to Processing Elements that consume and emit events. The focus of S4 is on scalable and fault tolerant processing of massive numbers of events.

Finally, [27] presents work that uses stratums as a way to achieve modularization and distribution. An event processing network (EPN) dependency graph models event processing agents (EPA) and relationships among them. The stratification process assigns each EPA to one of the stratum levels and finally to computing nodes. The query processing in WS-Aggregation builds on this work: we determine the EPN from the aggregation query in which the processing logic of the EPAs is given by the window queries of the eventing inputs. The stratum levels in our case are built in an analogous way by determining which inputs have no unfulfilled data dependencies and can be assigned to an aggregator for processing.

VIII. Conclusion

We have presented a platform for active event-based aggregation of Web data. The active query model utilizes XQuery window clauses, and provides a language extension to model data dependencies between event streams or other query inputs. The system is designed for scalability and distributed query execution, and allows easy deployment in the Cloud. As part of our ongoing work, we are investigating advanced techniques for optimized load distribution, bundling multiple queries on shared event buffers, and further adoption of Autonomic Computing [28] concepts to the platform's control loops. Furthermore, to account for often recurring CEP patterns, we envision the creation of a domain-specific language (DSL) as a light-weight abstraction of XQuery clauses.

References

[1] C. Schroth and T. Janner, "Web 2.0 and SOA: Converging Concepts Enabling the Internet of Services," *IT Professional*, vol. 9, no. 3, 2007.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, 2007.

[3] H. Taylor, A. Yochem, L. Phillips, and F. Martinez, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*, 1st ed. Addison-Wesley Professional, 2009.

[4] W3C, "Web Services Eventing (WS-Eventing)," 2006. [Online]. Available: http://www.w3.org/Submission/WS-Eventing/

[5] ——, "XQuery 1.0: An XML Query Language," 2007. [Online]. Available: http://www.w3.org/TR/xquery/

[6] I. Botan, D. Kossmann, P. M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius, "Extending XQuery with window functions," in *VLDB*, 2007, pp. 75–86.

[7] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Co., 2010.

[8] W. Hummer, P. Leitner, and S. Dustdar, "WS-Aggregation: Distributed Aggregation of Web Services Data," in *Symp. On Applied Comp.*, 2011.

[9] R. Sayre, "Atom: the standard in syndication," *Internet Computing, IEEE*, vol. 9, no. 4, pp. 71 – 78, 2005.

[10] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient Pattern Matching Over Event Streams," in *ACM SIGMOD*, 2008, pp. 147–160.

[11] P. M. Fischer, A. Garg, and K. S. Esmaili, "Extending XQuery with a pattern matching facility," in *Int. XML Database Sym.*, 2010, pp. 48–57.

[12] S. Robison, "The Next Wave: Everything as a Service," http://hp.com/hpinfo/execteam/articles/robison/08eaas.html, 2008.

[13] W3C, "Web Services Addressing (WS-Addressing)," 2004. [Online]. Available: http://www.w3.org/Submission/ws-addressing/

[14] ——, "XQuery Update Facility 1.0," 2011. [Online]. Available: http://www.w3.org/TR/xquery-update-10/

[15] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[16] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed Complex Event Processing with Query Rewriting," in *DEBS*, 2009.

[17] M. T. Maybury, "Generating Summaries From Event Data," *Int. J. on Information Processing and Management*, vol. 31, pp. 735–751, 1995.

[18] G. Cugola and A. Margara, "TESLA: a Formally Defined Event Specification Language," in *DEBS*, 2010, pp. 50–61.

[19] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Advanced Event Processing and Notifications in Service Runtime Environments," in *DEBS*, 2008.

[20] E. Mulo, U. Zdun, and S. Dustdar, "Monitoring Web Service Event Trails for Business Compliance," in *SOCA*, 2009, pp. 1–8.

[21] L. Zeng, H. Lei, and H. Chang, "Monitoring the QoS for Web Services," in *ICSOC*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 132–144.

[22] Pixware, "Qizx, a fast XML database engine fully supporting XQuery," http://www.xmlmind.com/qizx/.

[23] S. Abiteboul, O. Benjelloun, and T. Milo, "The Active XML project: an overview," *The VLDB Journal*, vol. 17, pp. 1019–1040, August 2008.

[24] M. Altinel and M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *VLDB*, 2000, pp. 53–64.

[25] I. Miliaraki and M. Koubarakis, "Distributed Structural and Value XML Filtering," in *DEBS*, 2010, pp. 2–13.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *ICDM Workshops*, 2010, pp. 170–177.

[27] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion, "A stratified approach for supporting high throughput event processing applications," in *DEBS*, 2009, pp. 1–12.

[28] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41 – 50, Jan. 2003.